Data Structure & Algorithm for Combination Tree To Generate Test Case

Ravi Prakash Verma[1], Bal Gopal[2] and Md. Rizwan Beg[3]

[1] Department of Computer Science and Engineering, Integral University Lucknow, Utter Pradesh, 226026 India

[2] Department of Computer Applications, Integral University Lucknow, Utter Pradesh, 226026 India

[3] Department of Computer Science and Engineering, Integral University Lucknow, Utter Pradesh, 226026 India

Abstract
The combinations play an important role in software testing. Using them we can generate the pairs of input parameters for testing. However until now we have the tabular representations for combination pairs or simply the charts for them. In this paper we propose the use of combination trees which are far easier to visualize and handle in testing process. This also gives the benefits of the remembering the combination of input parameters which we have tested and which are left, giving further confidence on the quality of the product which is to be released.

Keywords: *Software testing, combination trees, Data structures, algorithm*

1. Introduction

The software testing is one the most important activity in the SDLC [4]. It authenticate whether the software being developed solves the intended purpose or not [2]. "Software systems continuously grow in scale and functionality" [1]. Therefore large size and complexity of software can introduces more error, bugs and faults, in this situation testing becomes more important to uncover errors, bug & faults before software is actually put to use. Software testing also confirms that software being developed as per requirements [5]. At present it is mostly done manually and the test cases are written by the tester, it is a manual activity [3] [6]. This is most error prone area as important path or case may be missed out by the tester [3]. The testers develop test cases on the basis of the combinations of value of input parameters taken one at a time, these test cases are represented in the tabular form. It becomes difficult to remember that all the combination have been listed out or not. Further it difficult to visualize that whether we have covered all input parameters decisions that can be taken by the user. The combination trees can show the decision or action taken by the uses in a sequence which is very important for the software developer and tester to prove the robustness of the software system being developed. Testing done on the bases of combination trees [7] ensures that we are covering every possible action that can be taken by the user or at least can ensure that software system performs correctly if valid condition & action are chosen. In this paper we present a formal data structure and algorithm to generate the combination trees from the set of elements represented in array.

2. Proposed work

The number of *k*-combinations from a given set *S* of *n* elements (distinct and no repeating) is often denoted by ^nC_k which is $\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n(n-1)\ldots1}{k(k-1)\ldots1}$. When k > n/2 then $\begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} n \\ n-k \end{pmatrix}$ for $0 \leq k \leq n$. The total number of combination from n distinct elements is = $^nC_0 + {}^nC_1 + {}^nC_2$ + … $^nC_{n-1} + {}^nC_n$ which is 2^n or $\sum_{i=0}^{i=n} {}^n_iC$. As we see that $^nC_0$ represents null or empty elements in the set, however this is not the case in testing as this represents the case where we do not have input, so ignoring this we have = ${}^{n}C_1$ + ${}^{n}C_2$ + … ${}^{n}C_{n-1}$ + ${}^{n}C_n$ which is 2^n -1 or $\sum_{i=1}^{i=n} {}^{n}_{i}C$. For example if we have S = {a, b, c}, n = |S| = 3. The total number of combination are given by = ${}^{3}C_1 + {}^{3}C_2 + {}^{3}C_3$ = 3 + 3 + 1 = 7 or 2^3 -1 = 8. The sets are given as follows {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}. If we want to generate combination tree for this set S we start with root, which represents the null or empty set initially, this is level zero. For making level 1 then we add all the distinct elements from the set and make root as their parent not that the number of levels in the combination tree are n+1 where n represents number of distinct nodes, the level start from 0, 1, 2, … , n. After that we add (make child) next element from the set S higher in some order preferably lexicological order to the first child at level 1, once these are fixed we select next child and here also we take element higher in lexicological order and add them until all elements in the set are exhausted. Then the same is repeated until all levels are occupied. The combination tree representation of the combination just generated is shown by the tree in figure 1.

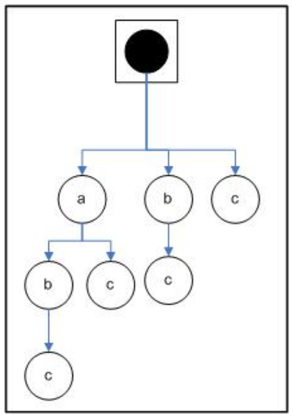

Figure 1. Showing combination tree

The sets and its element can be represented as conditions or the input given to the software module. The combination trees connects these conditions and input values and we can it imitate the users action and choices if follow a particular part in the combination tree. It gives complete listing of action that users can do. The testers can follow a particular path and decide what software should be doing under a situation and decide whether the software module should pass or fail on particular path.

Now we formalize the above method into algorithm and give its supporting data structures. First of all we need a structure to represent a tree node having data, pointer to parent and pointers to child, which is given as follows.

```
struct node { char [ ] value ;
              int iChild;
               structure node *Parent;
               structure node *Child [Max];
             }
```

The Max can take value of N, where N is the number of elements in the set S represented by array. Next we define auxiliary function to create a node, which is given as follows.

```
struct node * Root = NULL;

node * makeNode(char data, int nC, int i)
 { node * temp = (node *) malloc(sizeof(node));
   if (i = = 1)
    { temp->value = data; }
   else
    { temp->value = 'R'; }
   temp->Parent = NULL;
   temp->iChild = nC;
   for (int j = 0; j < temp->iChild; ++j)
    { temp->Child[j] = (node *) malloc(sizeof(node));
      temp->Child[j] = NULL;
    }
   return (temp);
 }
```

The root of the tree is the special node having no data but it has pointers to its children and it parent field is set to NULL. The auxiliary function to create root node is called with “nC” as “Max” and “I” as “0”.

We need and auxiliary array or list to store the nodes at given level which server as parent to the child below the current level. The linked list representation of pointers to nodes is used to store intermediate result. One of the advantages provided by this storage is that it avoids back tacking and traversal. The size of this pointer array first increases then it starts to reduce and finally reduces to zero size in length. This happens because in $\sum_{i=1}^{i=n} {}^{n}c_i$, ${}^{n}c_i$ equals ${}^{n}c_{n-i}$, which is $2^{(n-1)}$ -1. For this we define node structure PPNode and “addParentPointer” auxiliary functions to add nodes in the list and “removeNodeFromHead()” to delete the added nodes from the beginning in FIFO order. The PPHead & PPTail are pointers to handle the list. These are as follows.

```
struct PPNode { struct  node * N;
                struct node * next;
                };
```

```
struct PPNode  * PPHead = NULL;
struct PPNode  * PPTail = NULL;

void addParentPointer(node * n)
 { PPNode * temp = (PPNode*) malloc
                         (sizeof(PPNode));
   temp->N = n;
   temp->Next = NULL;
   if (PPHead == NULL && PPTail == NULL)
    { PPHead = temp;
      PPTail = PPHead;
      Root = n;
    }
   else
    { PPTail->Next = temp;
      PPTail = temp;
    }
}

void removeNodeFromHead()
 { PPNode  * temp = (PPNode *) malloc
                         (sizeof(PPNode));
   temp = PPHead;
   if (PPHead != NULL && PPHead->Next !=NULL)
    { PPHead = PPHead->Next; }
   else
    { PPHead = NULL; }
   free(temp);
 }
```

Another auxiliary function is used to set the index value such that the element in the Array is greater than its parent in terms of lexicographical order, this is given as follows.

```
int setIndex(PPNode * T)
 { int j = 0;
   char x = T->N->value;
   for (int i = 0; i < Max; ++i)
    { if (x == Array[i])
       { j = i;
             i = Max;
       }
    }
    return (j+1);
 }
```

Last we need an array to store the distinct elements and Max is the number of elements in array. To start creating the tree we set head & tail of the linked list to NULL and root of the tree to NULL. Finally the "createCombinationTree" function creates the combination tree and is given as follows.

```
void cCTree(int _Max)
{
1.  addParentPointer(makeNode(NULL, _Max, 0));
2.  i = 0;
3.  while (PPHead != NULL)
4.   { j = 0;
5.    while ( i < _Max)
6.     { node * n =  makeNode(Array[i], _Max-i-1, 1);
7.         n->Parent = PPHead->N;
8.         PPHead->N->Child[j] = n;
9.         addParentPointer(n);
10.        i = i + 1;
11.        j = j + 1;
       }
12.    j = 0;
13.    removeNodeFromHead();
14.    i = setIndex(PPHead);
     }
}
13.       temp = temp→next;
14.       removeNodeFromHead();
15.       i = setIndex(temp);
        }
}
```

3. Proof and analysis

For a set of elements S containing n elements a combination tree can be generated, where the elements are distinct and repetition in generated combination are not allowed. In order to prove that combination tree algorithm generates all the combination successfully and the loops terminate and the algorithm halts, we use the loop invariance method [8], which is given as follows:

3.1. Proof

Initialization: Prior to the beginning of the loop the link list "ParentPoiunterNode" is empty.

Maintenance: To see that, at each iteration maintains the loop invariance we start with the root, that is the first node that is added, i is initialized to zero and the immediate child of the root gets insert into the tree as well as in the list. Once the insertion is complete we remove the first node root from the list and this time the i gets the new value 1 and this time also the list is not empty but contains the new roots at next level. Once the value of i is exceeds the maximum number of elements then new node are not being added to the list instead they are removed from the head.

Termination: At termination we see that node are removed one by one as i get the value always higher then maximum, therefore nodes are removed one by one and finally the list becomes empty.

3.2. Complexity Analysis

To establish the upper bound in the proposed algorithm, to represent the worst case run time, we have to do approximation at various places in order to simply the analysis. We start by measuring the upper bound of various auxiliary procedure used and them using them in the proposed algorithm for final rough estimation. The function "makeNode(data)", "makeRootNode()" and "setIndex(struct PPNode * T)" have the complexity of $O(n)$. The complexity of "setIndex(struct PPNode * T)" is the approximate value as the complexity decreases as the node starts taking it places in the tree since first time it get called it takes n units of time, second time it takes n-1 units of time and finally it stats taking $O(1)$ time. The functions void "addParentPointer(struct * node)" and void "removeNodeFromHead()" take $O(1)$ time. for the algorithm "createCombinationTree" we start with step 1 which takes $O(n)$ time, step 2 takes $O(1)$ time, step 3 has a loop which executes taking $({}^nC_1 + {}^nC_2 + \dots {}^nC_{n-1} + {}^nC_n = 2^n - 1)$ $O(2^n)$ time, step 4 take $O(2^n)$ time, step 5 is loop taking maximum time of $O(n2^n)$, 6 takes $O(n)$ time step 7-12 take $O(1)$ individually & they are in two loops therefore take total time of $O(n2^n)$, step 13 take $O(1)$ and finally step 14 takes total time of $O(n2^n)$. Summing up the total time of each step we get

$= O(n) + O(1) + O(1) + O(2^n) + O(2^n) + O(n2^n) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(n2^n).$

$= O(n) + 10O(1) + 2O(2^n) + 2O(n2^n)$

Ignoring constant we have

$= O(n2^n) + O(2^n) + O(n)$

$= O(2^n(n+1)) + O(n)$

Ignoring lower order terms we have

$= O(n2^n)$

So the approximate worst case complexity of the creating combination tree is $O(n2^n)$.

4. Conclusion & future work

The combinations can be generated by reading the vertices and follow leading edges as path to other vertices, when we start from a root & descend to child, the combination pair is, all node encountered while descending from root to the leaves of the tree. There fore to generate combination pair having 2 elements we have to descend to depth of two. The root of the tree is at depth zero, so we follow every path from the root to depth of two. This is how we have generated the combination tree which assumes that there are distinct elements in the set S having n number of elements. We have generated non repeating combination with over all complexity of $O(n2^n)$. For the future work we should try to establish more accurate upper bound on the algorithm and also reduce the fixed space take by each node as the number of child of a node in the combination tree varies, these are maximum for the roots & decrease when we descend in the tree, therefore memory requirement drops and also the number of sub paths decrease.

References

[1] Kaschner, K., Lohmann, N., "Automatic Test Case Generation for Interacting Services". In Proc. of ICSOC 2008 Workshops. Volume 5472 of Lecture Notes in Computer Science. (2009)

[2] Tony Hoare, "Towards the Verifying Compiler", In The United Nations University / International Institute for Software Technology 10th Anniversary Colloquium: Formal Methods at the Crossroads, from Panacea to Foundational Support, Lisbon, March 18–21, 2002. Springer Verlag, 2002.

[3] Robert V. Binder, "Testing Object-Oriented Systems: Models, Patterns, and Tools", Addison Wesley Longman, Inc., 2000.

[4] S. S. Riaz Ahamed, " Studying the feasibility and importance of software testing: An Analysis", International Journal of Engineering Science and Technology, Vol.1(3), 2009, 119-128.

[5] Glenford J. Myers, "The Art of Software Testing", Second Edition, John Wiley & Sons, Inc.

[6] B. Beizer "Software Testing Techniques", Van Nostrand Reinhold , 2nd edition, 1990.

[7] Jaroslav Nesetril, "ASPECTS OF STRUCTURAL COMBINATORICS (Graph Homomorphisms and Their Use)", TAIWANESE JOURNAL OF MATHEMATICS Vol. 3, No. 4, pp. 381-423, December 1999

[8] Thomas H Cormen, Clifford Stein, Ronald L Rivest, Charles E Leiserson, "Introduction to Algorithms (2001)", McGraw-Hill